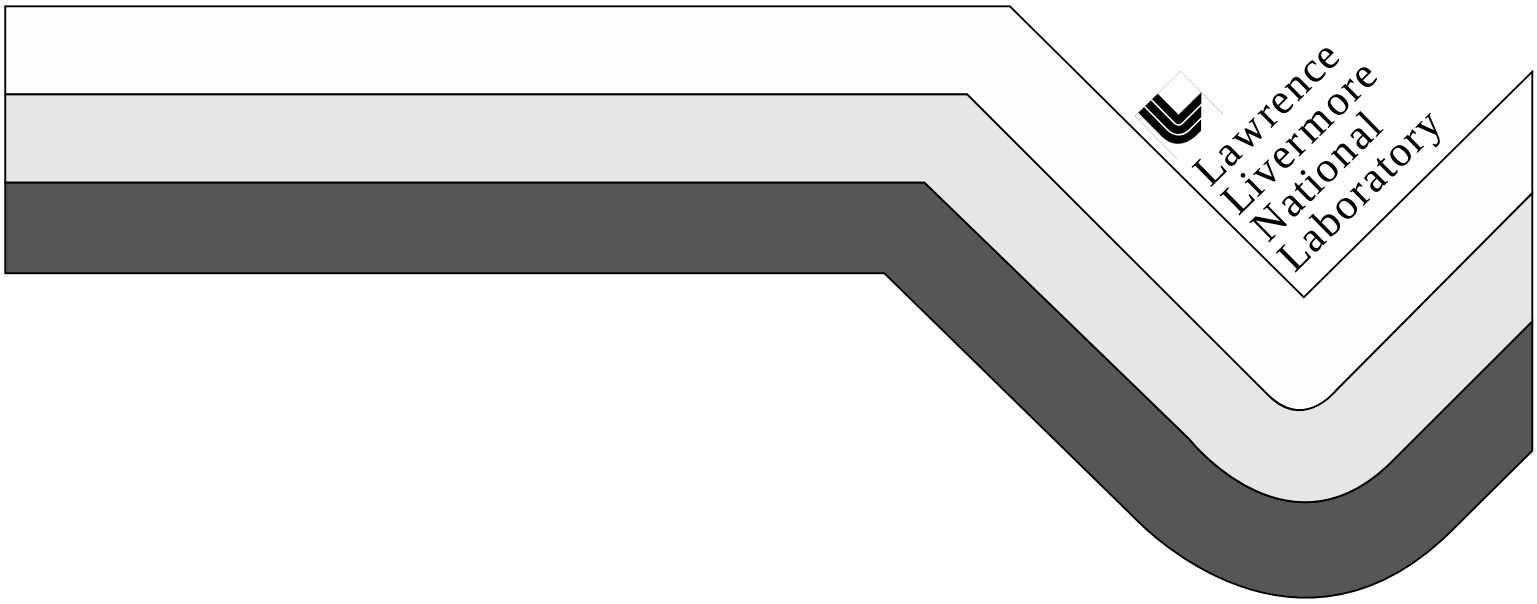
UCRL-MA-119225

Oil Shale Process Model (OSP) Code Development Manual

C. B. Thorsness and D. F. Aldis

December 6, 1994



Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

Oil Shale Process Model

(OSP)

Code Development Manual

C. B. Thorsness and D. F. Aldis
Lawrence Livermore National Laboratory
P.O. Box 808, L-365
Livermore CA 94550

TABLE OF CONTENTS

PREFACE i
INTRODUCTION 1
CODE STRUCTURE 1
UNIT OPERATION MODULES 2
OSP FILES 4
    Source Code Files 4
    I/O Files 5
    OSP Files and Module Common Space Structure 5
UNIT OPERATION MODULE DATA CONTROL AND COMMON SPACE STRUCTURE 6
PROCESS STREAM CHARACTERISTICS 7
    Process Stream Common Space Structure 8
PROCEDURE FOR ADDING A UNIT OPERATION MODULE 9
ADDING A NEW INPUT PARAMETER FOR A UNIT OPERATION MODULE 18
ADDING A NEW CHEMICAL SPECIES 18
ADDING A NEW AUXILIARY SOLID PROPERTY 19
ADDING A NEW CHEMICAL REACTION 19
    Routines in "service.f" File 21
    Routines in "props.f" File 24
    Routines in "reaction.f" File 25
    Routines in "combine.f" File 25
    Routines in "plug.f" File 25
    Routines in "well_mix.f" File 26
    Routines in "binary_io.f" File 26
SPECIFIC MACHINE DEPENDENCIES IN THE OSP CODE 26
    Precision Options for LSODE 27
REFERENCES 27

PREFACE

The Oil Shale Process (OSP) model has proven to be a useful tool for the analysis of the steady-state operation of Lawrence Livermore National Laboratory's Hot-Recycled-Solids 4 tonne-per-day Pilot Retort (4TU-Pilot). This manual has been developed to guide a user through the development of source code modifications to the OSP code and to assist a user in implementing the code on a computer different than the HP workstations that have been used for its development.

The OSP code has been developed for generalized chemical process simulation of the HRS (Hot-Recycled-Solid) oil shale retort. Several of the common unit operations encountered in industrial chemical process design are included in the OSP code, including packed and fluidized bed reactors, splitting and mixing operations, and a dilute flow lift-pipe. A set of chemical species and reactions related to oil shale processing are available for incorporation into a given simulation.

This manual will guide an engineer / programmer through the code structure, logic and data structures or data files used by the OSP code. Two other manuals complete the documentation for the OSP code, the "Oil Shale Process Model (OSP) Theory Manual" and the "Oil Shale Process Model (OSP) User's Manual." These manuals should be referenced to find out the why and how of OSP.

INTRODUCTION

The computer code OSP has been developed to allow a generalized-steady-state-numerical simulation of the Hot-Recycled-Solids (HRS) Oil Shale Process to be constructed. The HRS process is being developed at Lawrence Livermore National Laboratory (LLNL) for the above ground retorting of oil shale. The OSP code helps research staff to critically analyze operation of LLNL's 4TU-Pilot. It can also be used to test the consistency of the laboratory data as well as the pilot scale results.

This manual describes the computer programming details of the OSP code. This manual should be consulted if the code is to be moved from a HPUX operating system to any other operating system in order to identify specific machine dependencies. It should be used to assist an engineer / programmer in updating existing coding, in adding new functionality, or in modifying the code's structure. Two other manuals have been developed to describe other aspects of the OSP code. The "Oil Shale Process Model (OSP) Theory Manual" describes the theoretical basis for the assumptions, equations, and numerical methods used in the OSP code. The "Oil Shale Process Model (OSP) Users Manual" describes the requirements for the user generated input file; its organization, format, and an example of its content.

The OSP code uses information contained in a user supplied input file to select the order, number and type of computational unit operations used in a simulation. The chemical species and the number and type of chemical reactions occurring in a unit operation are also specified in the input file as are initial estimates for processing conditions, stream flow rates, and chemical compositions.

CODE STRUCTURE

The OSP code has been structured in a generalized manner so that any number of unit operations can be added to a simulated process in a convenient and efficient manner. The type of units used in a simulation can be selected from those units listed below by modifying the user supplied input file. The number and type of chemical reactions and the chemical reaction stoichiometry can be selected by the user from the set of reactions presently available in OSP by modifying the user supplied input file.

The code is structured so that the main routine acts as a controller for the code. It allows lower level unit operation modules to perform initialization, computation and final summary printing and reporting steps.

The unit operation modules, which are implemented through a primary subroutine (which calls a series of secondary subprograms to carry out the required tasks), are called by the main controller, routine. The type of computational unit operation modules supplied in the OSP code have been selected to allow one to simulate the HRS

process pilot plant. The principle unit operations used in the HRS process are listed below:

° Packed Bed Pyrolyzer

° Lift-Pipe for Solids Transport and Combustion

° Delayed-Fall Combustor

° Fluidized Bed for Solids Mixing and Reaction

° Splitting and mixing operations

Each of these unit operations are simulated with one or more computational modules. The modules consist of an initialization section, a computational section, and a summary-report generation section. The chemical reactions considered in specific computational unit operation module are specified in the input file.

The unit operation modules are connected to each other and to specified system input and output streams with one or more process streams. OSP uses three types of streams, gas, liquid and solid. A gaseous process stream is characterized with a pressure, molar flow rate, composition and temperature. A solid process stream is characterized with a mass flow rate, composition, temperature, auxiliary properties and particle size. A discretized particle size distribution can be simulated by using several solids streams, one for each size class. Although OSP's primary focus are on interactions between solid and gas species a liquid streams are also available and can be used in a limited number of ways. A liquid process stream is characterized with a mass flow rate, composition and temperature. Initialization of the independent process streams is performed using information contained in the input file.

UNIT OPERATION MODULES

Several unit operation modules are presently incorporated in the OSP code. Each module has an initialization section, a calculation section, and a summary-report section. During the initializing section, parameters are assigned to variables used in the module. In many cases a parameter is assigned a default value if no parameter is specified. Parameters used in a computational module listed in the users supplied input file should have the format given below:

PARAMETER1 = VALUE1 PARAMETER2 = VALUE2

The name of a parameter, e.g... PARAMETER1, is followed by an equal sign, which is then followed by the value of the parameter to be used in the simulation (spaces around the equal sign are optional). Several parameters can be assigned on a single line in the input file. A default value may be used if a parameter is not supplied. If an invalid

name of a parameter is listed, the program echoes the line on which the error occurred. Each individual module is responsible for generating these error messages. This format is similar to the NAMELIST feature supported by some FORTRAN compilers, but it is not NAMELIST. This convention has been implemented in standard FORTRAN77, without use of the NAMELIST feature.

The following is a list of the computational modules presently in the OSP code along with a very brief description: (A more detailed description, which includes a list of the input parameters for each module, is given in the Oil Shale Process Model (OSP) Users Manual.)

ATTRITION - Simulates the change in particle size that occurs in the processing of oil shale.

BALANCE - Performs overall atomic and energy balances as well as kerogen-oil-coke balances useful in analyzing overall oil generation performance.

CNTR_CURRENT - Simulates a counter-current gas/solid flow reactor.

CO_CURRENT - Simulates a dilute phase co-current plug flow type reactor.

CSTR - Simulates a continuously stirred tank reactor.

ERROR - Used to define auxiliary error criteria to be used in convergence of recycle loops.

FLUID_BED - Simulates a two phase fluidized bed reactor which includes a bubble phase and an emulsion phase .

LIFT_PIPE - Simulates a dilute phase lift pipe using a plug-flow assumption with provisions for computing gas/solid slip velocities.

MERGE_STRMS - Combines multiple input streams into a smaller number of output streams all assumed to be in thermal equilibrium (i.e. same temperature).

PACKED_BED - Simulates a dense phase co-current plug-flow reactor .

PASS_THRU - Passes input streams straight through to output streams with no alteration which can be useful in loop isolation and renaming streams.

PHASE_CHANGE - Allow transfer of material between gas and liquid phases.

PROP_TAB - A service module that prints out a table of all species and mixture properties over a range of temperatures.

RELAX - Provides for solid stream updating which blends old and newly computed values of solid stream parameters which can be helpful in reaching a converged solution in cases with recycle loops.

SPLIT_STRMS - Allows single streams to be split into two output streams based on fractional splitting or specification of a desired flow rate.

STOICH - Determines the stoichiometry of a given reaction an allows the heat of reaction over a selected temperature range to be tabulated.

STOICH_REACT - Simulates reactions in which extent of reaction is defined by parameters supplied by the user.

OSP FILES

Several files are associated with OSP. These include source code files and files used during the execution of the code.

Source Code Files

The source coding for OSP is contained in files with the suffix ".f". A common space is typically established for each module. An ASCII version of the common space for a module is in a file which has a suffix ".i". The first and last variable locations in the common block are identified as integers and are used to located the start and end position in memory of the particular common block. The name of the starting location variable has a suffix of "_start" and the ending location variable has a suffix of "_end".

The current source files for OSP, excluding specific module files, are:

Makefile - A make file used to compile and load OSP

binary_io.f - FORTRAN source file for routines used in reading and writing the ".mod" and ".strm" files.

combine.f & combine.i - FORTRAN source code for routines used in combining streams.

lsode.f & lsode.i - FORTRAN source code for the LSODE numerical integration package.

osp.f - FORTRAN source code for the primary OSP controlling routines.

osp_control.i - A FORTRAN include file containing common areas associated with program control and timing.

osp_flags.- A FORTRAN include file containing a common area associated with the $FLAGS- variables.

osp_input.doc - An ASCII text file which contains detailed information on constructing an input file for OSP.

osp_modules.i - A FORTRAN include file containing commons associated with module information useful to the overall control routine.
osp_params.i - A FORTRAN include file containing parameter statements and the global common area.
osp_props.i - A FORTRAN include file containing species property variables.
osp_rate.i - A FORTRAN include file which contains a common area associated with the RATE_GEN routine as well as a parameter statement defining the maximum number of rate expressions define.
osp_streams.i - A FORTRAN include file which contains commons associated with stream variables.
plug_flow.f & plug_flow.i - FORTRAN source code which implements the plug-flow model.
props.f - FORTRAN source code containing species property related routines as well as the DATA statements used to default species properties.
reaction.f - FORTRAN source code containing a series of routines used to define reaction stoichiometry and kinetics.
service.f - FORTRAN source code for many service related routines.
tab.f & tab.i - FORTRAN source code for the post processor tabulating code TAB.
well_mix.f & well_mix.i- FORTRAN source code which implements the well-mixed model.
work.i - A FORTRAN include file which declares workspace common used by the LSODE package.

I/O Files

Four I/O files are used by OSP during execution. These files include the input file which has the suffix ".inp", the ASCII output file which has the suffix ".dat", the stream file which has the suffix ".strm", and the binary-common-space-storage file which the suffix ".mod". The stream file can be used to restart a OSP calculation at a previously determined stream state or by the TAB to print selected computed information.

The binary-common-space-storage file is used to hold binary versions of all of the module common spaces for a particular simulation. The values of input parameters for each separate use of each module is stored in this file. The order of the common spaces in this binary file is the order that the modules were encountered in the input file. This file is written and read by each individual module program unit. The main OSP control routines handle the proper positioning of the file to allow proper access by each module.

OSP Files and Module Common Space Structure

Several variables have been included in the OSP code to allow communication between modules and the main controller routine. Some of these variables are set by the control routine and some are set by each module. These variables have the name prefix MOD.

The list of mod-variables, in this list SEQ is the sequence number of the module assigned by the control routine for each individual invocation of a module, includes the following:

MOD_ERROR(SEQ) - the module error defined by each module for use by the control routine to determine if a computation loop has converged. Set by each module.

MOD_NAME(SEQ) - the name of the module. Set by the controller.

MOD_TAG(SEQ) - the tag number of the module. Set by module form data in input file.

MOD_VERSION(SEQ) - the version number of the module. Set by coding in each module.

MOD_DESC(SEQ) - the description of the module. Set by module form data in input file.

MOD_TYPE(SEQ) - used to identify modules that are only called once, such as STIOCH. The defined types are INIT meaning a module only needs to be called during initialization sequence and CALC meaning this is a normal computational module which needs to be called for initialization, calculation and final output. Set by each module.

MOD_STRM_TYPE(SEQ,STRM_NUM) - the type of stream associated with the STRM_NUM'th stream used by the module. The currently defined stream types are GI, LI, and SI for entering gas, liquid and solid streams respectively, and GO, LO and SO for exiting streams. A type GB is also defined by the FLUID_BED module to designate an internal bubble stream. Each module should conform to this naming scheme since general purpose routines used to print output rely on this flag to properly function. Set by each module.

MOD_STRM(SEQ,STRM_NUM) - the number of the stream outside of the module corresponding to STRM_NUM'th internal stream. Set by each module.

MOD_GROUP(SEQ) - the group number of the module. Set by module form data in input file.

MOD_RECORD(SEQ) - the location of this specific module's common file in the ".mod" file . Set by the controller.

MOD_FILE_RECORD(SEQ) - the number of records in the ".mod" file used by the module. Set by each module.

MOD_PRINT(SEQ) - flag used to indicate if information from the module is to be printed to the ".dat" file during an output request. Set by the controller.

All of these variables are stored in a common block defined in "osp_modules.i".

UNIT OPERATION MODULE DATA CONTROL AND COMMON SPACE STRUCTURE

Each unit operation module has a set of parameters that are either assigned by default or read in from the input file. Each set of parameters is stored, by the module that uses

it, during it's initialization step into a common space for each module and in a BINARY formatted file which has the suffix ".mod". The ".mod" file will contain a copy of the common spaces used by all of the modules used in a simulation. A unique index number is assigned by the main control program at the beginning of the initialization process for each module common space. This index number is the order in which the modules were encountered in the input file. If a particular type of module is used more than once in a simulation, each common block for each implementation of the module will be stored independently. When the calculational step of the OSP code begins, the common blocks in the ".mod" file will be addressed separately. The information in this file is only modified during the beginning initialization step. This approach has been developed to allow a particular unit operation to be used several times in a simulation, with different input process parameters.

When a unit operation module begins a calculation step, it must first read in the parameters stored in the common block, from the ".mod" file. Because parameters in the ".mod" file are not altered in the calculational step, it is not necessary to change the file at the end of a calculational step. This is different than the stream file that will be described in the next section. Arrays stored in the stream file do change as the calculational process proceeds.

PROCESS STREAM CHARACTERISTICS

In the OSP code, unit operation modules are connected with process streams. Three types of process streams are used, gas streams, solid streams and liquid streams. Each of the streams have stream variables associated with them. Each of the three stream types has flow rate, temperature and composition variables. The units for the flow rate of a gas stream is moles per second and the units for the flow rates of solid or liquid stream is kg per second. The temperature is maintained in Kelvin units and the composition of the gas streams are in mole fraction units while those for liquids and solids are weight fractions. Gas stream variables also include a pressure in Pascals.

The solid streams have several other defined stream variables. One group are auxiliary property variables. These variables are treated like compositions during combination and splitting processes. Three auxiliary properties are currently defined. The first is the original char content of the solid. This value is used by the attrition and char combustion models. The second is the original kerogen content of the solid. This property is used in estimating oxygen diffusivities in the char combustion model. The third property is the original FeS2 content also used in combustion models.

The particle size and porosity are also part of a solid stream characterization. If a distribution of solid size classes is required, then the distribution should be discretized into several separate streams each with appropriate particle size and flow rate.

The solid stream auxiliary property parameters have been implemented in a general fashion to allow new properties to be added. The number of these properties is defined in the "osp_params.i" file by the parameter MAX_PROP_SOLID. Coding within OSP

will handle the passing of these properties from input to output streams and will combine streams with different values using rules equivalent to those used in computing compositions. When these properties are printed out in the ASCII output file a label defined in the "outp_gen" subroutine located in source file "service.f" is used. Addition of a new property should include an appropriate label. Of course, a new property will only have an influence on the course of computations if new coding is added which makes explicit use of this property variable. It is also necessary that any new module properly pass on these parameters from input streams to output streams regardless of whether or not they are used explicitly in the module.

Process Stream Common Space Structure

Information about the streams in the OSP code are contained in three process stream common blocks, one for gases (called GASC), one for solids (called SOLIDC), and one for liquids (called LIQC). All three common blocks are located in the file "osp_streams.i". The gas common block contains arrays for the molar gas flow rates in the streams, for the gas temperatures, for the gas species mole fractions, and for the molecular weight of the gas in the streams. The liquid common block contains a corresponding set of four arrays. The solid common block contains arrays for the mass flow rates in kg/sec, the temperatures of the solids in the streams, the solid species mass fractions, the diameter of the particles in each of the streams, the absolute density of the solid in the each of the streams, the porosity of the solid in the streams, and auxiliary properties.

During OSP initialization, the stream variables are initialized using information in the input file. ASCII strings are used in the input file to identify streams used by a module. An input stream might be called INFLOW. This name is then mapped to a sequential stream number. This sequential stream number is then used as a reference to the stream. As each unit operation module begins, a calculational step it will extract information from the stream common space for those streams entering the specific module. At the end of a calculational step, the unit operational module will update the stream common space with the most recent calculational results for the output stream variables.

As a programming convenience, the stream variables are generally copied into a set of variables that are local to the module and not in common with the stream variables. At the end of the calculational step the results are copied out into the variables in the stream common block.

The file that contains the stream variables has a name with a suffix ".strm". If a user needs to restart a calculation, the stream file can be used for initialization. Using a stream file for initialization can reduce computational time significantly.

PROCEDURE FOR ADDING A UNIT OPERATION MODULE

A unit operation modules in general consist of four parts; an initialization section, a calculation section, a ".mod" file read / write section and an ASCII output section. In general when adding a new module, all four of these parts should be included. Other conventions that have been adopted in the OSP code are described below. In the initialization section default parameter values are set and module input data is read from the input file. For most modules the section also includes coding to write the setup parameter values to the ASCII output file. The ".mod" file read / write section is invoked in the write mode at the end of the initialization to write out parameters values. It is also invoked at each calculational call to the module to read in the proper module parameters for that invocation of the module. When the controller decides that the solution for the overall process has converged each module is called so that appropriate output for the module can be written to the ASCII output file. The modules are signaled by the controller as to the type of action required by passage of an character variable which is set to either 'INIT', 'CALC' or 'OUTP'.

Parameters listed in the ".mod" file common space used by the module can only be modified during initialization, and not later. In general, a single common is used but a given module may use more than one common block.

Variables in module local common spaces need to be either assigned or recalculated at the beginning of each calculational step. This is necessary because a module can be used more than once in a simulation and the local common spaces can be changed each time the module is used. This means the only memory available to a module is through the stream common and the initial values set in the ".mod" file during initialization.

The procedure of adding a new unit operation module to the OSP code is described below. The PASS_THRU module subroutines are used as a coding example.

Two files are typically used by a module, the first file will contain the coding and the second include file contains common space definitions. Code files use a suffix of ".f" and include files a suffix of ".i". Special requirements for module common structure and use have been described above.

The coding file in general will contain several subroutines. The names of subroutines used in the file should have a distinctive 2 letter suffix, to avoid confusion between subroutines in other modules. The first subroutine shown in Fig. 1 controls the execution of other module routines. In the main routine, one of three steps will be initiated; initialization, calculation or output and reporting. The type of calculation to be performed by the module is passed into the module from the OSP main routine as an argument TYPE. This module's main routine will the call the appropriate module subroutines. The names of the routines used by the module to do these three types of actions should have the following prefixes; INIT, CALC, and OUTP. Before the calculational and output subroutines are called, a function with a prefix MOD_IO is

called. In this subprogram the common blocks for the module are copied from the ".mod" file into the common spaces to be used by the module.

```
      subroutine pass_thru(type,seq,ier)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c  PASS THRU
c
c  This module allows a gas, liquid and/or solid streams to be passed
c  through unchanged but renamed.
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_streams.i'
      include '../osp_modules.i'
c  formal params
      character type*(*)
      integer seq,ier
c  local parameters
      integer mod_io_ps


      mod_version(seq) = '1.0'

      if (type.eq.'INIT') then
         call init_ps(seq,ier)
         call outparams_ps(seq)
      elseif (type.eq.'CALC') then
         if (mod_io_ps(seq,'R',ier) .ne. 0) return
         call calc_ps(seq,ier)
      elseif (type.eq.'OUTP') then
         if (mod_io_ps(seq,'R',ier) .ne. 0) return
         call outp_ps(seq,ier)
      endif
c
      return
      end
```

Figure 1. Sample Module Main Subroutine

The first text after the beginning of the main routine for a module is normally a description of the module. Included in this description, are definitions of the input parameters and their default values. This is followed by a declaration of version number.

The INIT subroutine, shown in Fig. 2, usually begins with the assignment of default values for all of the parameters to be used by the module. Next, user specified values for the parameters are read in from the input file. Several functions and subroutines have been developed for reading data from the input file. One is available for scalar variables, for ASCII variables, for single subscripted arrays, and for double subscripted arrays. It is helpful to use an existing module, code file as a template to create a new module. At the end of the routine the initialized parameters contained in the module common block are written to the ".mod" file by the subprogram MOD_IO_PS.

```
c-----------------------------------------------------------------------

      subroutine init_ps(seq,ier)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c
c  Read input and initialize for calculations
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_streams.i'
      include '../osp_modules.i'
      include 'pass_thru.i'
c  formal params
      integer seq,ier
c  local parmas
      character line*132,key*80,line_in*132, strm_name*8, type*1
      logical value_flag, new
      integer read_line,loc,get_next_key,length,trimlen,parse,nstrm
      integer mod_io_ps, idum, i, first, last
      integer strm_map, strm_num, ngi,ngo,nli,nlo,nsi,nso
      real*8 value

      nstrm=0

c  read file for module parameters
      rewind (luin)
      do i=1, mod_record(seq)
         read(luin,'(a)',iostat=ier,err=900) line
      enddo
      line_in=line
      call casefold(line)
      loc=1
      length=trimlen(line)
c  this check is not necessary, mainly for debug
      loc=index(line,'$MODULE')
      if (loc.gt.0) then
         loc=loc+7
         if (loc.ge.length) then
            ier=-10
            goto 900
         endif
         if (parse(line,loc,key).le.0) then
            ier=-11
            goto 900
         endif
         if (parse(line,loc,key).le.0) then
            ier=-12
            goto 900
         endif
         if (key(1:20).ne.mod_name(seq)) then
            ier=-13
            goto 900
         endif
      else
         ier=-14
         goto 900
      endif
```

Figure 2. Sample Initialization Subroutine (continued)

```
      do while (ier.eq.0)

         do while (get_next_key(line,loc,key,value,value_flag,
     &                          first,last).gt.0)
            if     (key(1:4) .eq. 'DESC' ) then
               mod_desc(seq)=line_in(first:last)
            elseif (key(1:4) .eq. 'TAG ' ) then
               mod_tag(seq)=line(first:last)
            elseif (key(1:6) .eq. 'GROUP ' ) then
               if (.not.value_flag) goto 910
               mod_group(seq)=value

            elseif (key(1:3).eq.'GI ' .or. key(1:3).eq.'GO ' .or.
     &              key(1:3).eq.'LI ' .or. key(1:3).eq.'LO ' .or.
     &              key(1:3).eq.'SI ' .or. key(1:3).eq.'SO ') then
               type = key(1:1)
               strm_name = line(first:last)
               if (strm_map(type,strm_name,strm_num,new,ier)
     &             .lt. 0) goto 930
               nstrm=nstrm+1
               if (nstrm.gt. max_mod_strm) then
                  ier=-99
                  goto 900
               endif
               mod_strm(nstrm,seq)=strm_num
               mod_strm_type(nstrm,seq)=key(1:2)

            elseif (key(1:5) .eq. 'PRINT') then
               if (.not.value_flag) goto 910
               mod_print(seq)=value

            else
               ier=-6
               goto 900
            endif
         enddo
c
         ier=read_line(luin,line,ier,idum)
         if (index(line(1:5),'$') .gt. 0) goto 200
         line_in=line
         call casefold(line)
         loc=1
         length=trimlen(line)
      enddo
  200 continue
c
c  set in/out map
      ngi=0
      ngo=0
      nli=0
      nlo=0
      nsi=0
      nso=0
      do i=1,nstrm
         if     (mod_strm_type(i,seq) .eq. 'GI') then
            ngi=ngi+1
            gi(ngi)=mod_strm(i,seq)
```

Figure 2. Sample Initialization Subroutine (continued)

```
         elseif (mod_strm_type(i,seq) .eq. 'GO') then
            ngo=ngo+1
            go(ngo)=mod_strm(i,seq)
         elseif (mod_strm_type(i,seq) .eq. 'LI') then
            nli=nli+1
            li(nli)=mod_strm(i,seq)
         elseif (mod_strm_type(i,seq) .eq. 'LO') then
            nlo=nlo+1
            lo(nlo)=mod_strm(i,seq)
         elseif (mod_strm_type(i,seq) .eq. 'SI') then
            nsi=nsi+1
            si(nsi)=mod_strm(i,seq)
         elseif (mod_strm_type(i,seq) .eq. 'SO') then
            nso=nso+1
            so(nso)=mod_strm(i,seq)
         endif
       enddo

       ng=ngi
       ns=nsi
       nl=nli
c
c  check input data
       if (ngi. ne. ngo) then
          write(lutty,'(/
     &       '' *** # gas out streams not equal to number in.'')')
          ier=-21
          goto 920
       elseif (nli .ne. nlo) then
          write(lutty,'(/
     &       '' *** # liquid out streams not equal to number in.'')')
          ier=-22
          goto 920
       elseif (nsi .ne. nso) then
          write(lutty,'(/
     &       '' *** # solid out streams not equal to number in.'')')
          ier=-22
          goto 920
       endif

c  write common block to module file
       if (mod_io_ps(seq,'W',ier) .ne. 0) return
c
       ier=0
       return
c
 900   write(lutty,'(/'' *** Error'',i3,'' in init_ps.''
      &             /'' Line:''/a)') ier,line
       return
c
 910   write(lutty,'(/'' *** Error in init_ps, bad data.''
      &             /'' Line:''/a)') line
       ier = -9
       return
c
 920   write(lutty,'(/'' *** Error'',i3,'' in init_ps.''
      &             /'' Bad data'')') ier
```

Figure 2. Sample Initialization Subroutine (continued)

```
      return
c
 930  write(lutty,'(/'' *** Error'',i3,'' in init_ps.''
     &             /'' Problem with strm_map.'')') ier
      return
c
      end

c-----------------------------------------------------------------------
```

Figure 2. Sample Initialization Subroutine

The CALC subroutine, shown in Fig. 3, normally begins with the assignment of values to local variables that come from the ".strm" file. This is not necessary but is a convenient convention. In most cases, a lower level subroutine is used to perform the calculation, such as the well-mixed model or the plug-flow model. If one of these lower level routines are to be used an initialization step will be needed before the lower level module is called. After the calculations are performed, the modified values for the stream variables are assigned to the variables in the ".strm" common block. Note, no values of local variables are saved after the CALC subroutine returns.

```
c-----------------------------------------------------------------------
      subroutine calc_ps(seq,ier)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c
c  Do calculations
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_streams.i'
      include '../osp_modules.i'
      include 'pass_thru.i'

c  formal params
      integer seq,ier
c  local parameters
      integer i

      ier = 0

      do i=1,ng
         call stream_copy('GAS',gi(i),go(i),ier)
         if (ier .ne. 0) return
      enddo

      do i=1,ns
         call stream_copy('SOLID',si(i),so(i),ier)
         if (ier .ne. 0) return
      enddo

      do i=1,nl
         call stream_copy('LIQ',li(i),lo(i),ier)
         if (ier .ne. 0) return
      enddo

      mod_error(seq) = 0.0
```

```
      return
      end
c-----------------------------------------------------------------------
```

Figure 3. Sample Module Calculational Subroutine

Two subroutines are usually used for output. Parameter values that have either been obtained from the input file or assigned as default values are output using a subroutine with the prefix OUT_PARAMS, shown in Fig. 4, and values calculated by the module are output using a subroutine with the prefix OUTP, shown in Fig. 5. In most cases, the relevant results are in the stream data base and a utility routine is used to print out relevant stream information. However, in some cases other information specific to the module is calculated and printed out by the module.

```
c-----------------------------------------------------------------------

      subroutine outparams_ps(seq)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c
c  Write module parameters to luout.
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_modules.i'
      include '../osp_props.i'
      include 'pass_thru.i'
c  formal params
      integer seq

      write(luout,'(////1x,78(''*''))')
      write(luout,'(1x,''Mod: '',i2,'') '',a,a,''Ver:'',a)')
     &   seq, mod_name(seq), mod_desc(seq), mod_version(seq)
      write(luout,'(1x,78(''*''))')

      call connections(seq)

      write(luout,'('' '')')


      return
      end

c-----------------------------------------------------------------------
```

Figure 4. Sample Module Parameter Output Subroutine

```
c-----------------------------------------------------------------------
      subroutine outp_ps(seq,ier)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%c
c  Output results of calculations.
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_streams.i'
      include '../osp_modules.i'
c  formal params
      integer seq,ier
c  local parameters
      integer gflag(max_mod_strm), sflag(max_mod_strm)
      integer lflag(max_mod_strm)
c
c
c  only list non-zero components
      call set_flag(seq, gflag, sflag, lflag)
      call outp_gen(seq,'F', gflag, sflag, lflag)

      if (mod_print(seq) .eq. 1) call bal_overall(seq)

      return
      end

c-----------------------------------------------------------------------
```

Figure 5. Sample Module Calculational Output Subroutine

The binary I/O for the ".mod" file is performed in a subroutine that has a prefix of MOD_IO, as shown in Fig 6. The ".mod" file for a module contains a copy of the input parameters for an application of a specific module. By using a ".mod" file a module can be used multiple times in a process simulation. Each module has control of the reading and writing of its part of the ".mod" file. It may therefore use any number of records. However, the OSP controller routines must be informed of the number of records use by proper setting of the MOD_FILE_RECORD(SEQ) variable.

```
c-----------------------------------------------------------------------

      integer function mod_io_ps(seq,type,ier)
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c
c  Perform required io to binary module file.
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      include '../osp_params.i'
      include '../osp_streams.i'
      include '../osp_modules.i'
      include 'pass_thru.i'
c  formal params
      character type*1
      integer seq,ier
c  local parameters
      logical found
      integer i1, i2, nwords, write_words, iheader(9), iheader_rd(9)
```

Figure 6. Sample Module Binary Input-Output Subroutine (continued)

```
      integer read_words
      character header*36, header_rd*36
      equivalence (iheader,    header)
      equivalence (iheader_rd, header_rd)


      i2 = %loc(pass_thruc_end)
      i1 = %loc(pass_thruc_start)
      nwords = (i2 - i1)/4 + 1

      write (header,'('' Module '',i3,5x,a)')
     &      seq, mod_name(seq)

      if (type .eq. 'W' .or. type .eq. 'w') then
         if( write_words(lumod, iheader, 9, ier) .ne. 0)  goto 900
         if( write_words(lumod, pass_thruc_start, nwords, ier) .ne. 0)
     &      goto 900
         mod_file_record(seq) = 2

      else
         found=.false.
c  check of header not essential mainly for debug
         do while (.not.found)
            if( read_words(lumod, iheader_rd, 9, ier) .ne. 0)  goto 910
            if (header_rd .eq.  header) found=.true.
         enddo
         if (read_words(lumod, pass_thruc_start, nwords, ier) .ne. 0)
     &      goto 920
      endif

      ier=0
      goto 999

  900 write (lutty, '(/'' **** Problem writing to module binary file.''
     &               /''    seq='',i3,''   ier='',i6)') seq,ier
      goto 999

  910 write (lutty, '(/'' **** Problem positioning module binary file.''
     &               /''    seq='',i3,''   ier='',i6)') seq,ier
      goto 999

  920 write (lutty, '(/'' **** Problem reading module binary file.''
     &               /''    seq='',i3,''   ier='',i6)') seq,ier

  999 mod_io_ps=ier
      return
      end

c-----------------------------------------------------------------------
```

Figure 6. Sample Module Binary Input-Output Subroutine

The OSP main routine must be modified to let the OSP code know that a new module has been added. The main routine is in the file "osp.f" and the place to put the reference to the new module is in the CONTROL subroutine.

ADDING A NEW INPUT PARAMETER FOR A UNIT OPERATION MODULE

To add a new parameter to an existing module requires several steps. The new parameter needs to be given a name and included in the modules common area which is written to the ".mod" file. This common area is generally contained in the ".i" include file associated with the module. In the initialization section of the module coding a default value should be set and provisions made for allowing it to be set by user input through the ".inp" file. Also in the initialization routine the value of the parameter, which is finally set by default or by user input, should be written to the ASCII output file. The new parameter should be documented in the "osp_input.doc" file if appropriate. Finally the version number of the module should be changed.

ADDING A NEW CHEMICAL SPECIES

A new chemical species can be added to the OSP code by the steps outlined below.

The OSP code needs several physical and chemical properties to be defined for any new chemical species. For each of the three species type a name is required. The other required parameters vary somewhat depending on the type. The default values for all parameters are set in the block data area at the end of the "props.f" file.

For a gas, the following properties are needed: composition viscosity coefficients, heat of formation, heat capacity coefficients, and molecular volume. To specify composition the atoms / molecule of the five elements tracked by OSP, carbon, hydrogen, oxygen, nitrogen and sulfur need to be set. In addition, for each new gas specie, an entry needs to be made in the GAS_TO_LIQ_MAP table. If the gas is one which has a companion liquid phase defined then the value of the component GAS_TO_LIQ_MAP entry should be set equal to the sequence number of the liquid component into which it condenses. If there is no companion liquid phase defined then the value should be set to zero.

For a liquid, the following properties are needed: composition, heat of formation, and heat capacity coefficients. The composition is defining by setting weight fractions of the five elements tracked by OSP. In addition, for each new liquid specie, an entry needs to be made in the LIQ_TO_GAS_MAP table. If the liquid is one which has a companion gas phase defined then the value of the component LIQ_TO_GAS_MAP entry should be set equal to the sequence number of the gas component into which it evaporates. If there is no companion gas phase defined then the value should be set to zero.

For a solid, the following properties are needed: composition, heat of formation, absolute density, and heat capacity coefficients. The composition is defined by setting weight fractions of the five elements tracked by OSP. In addition a weight fraction for the inorganic carbon needs to be set. This later value is used for certain overall balances produced for convenience by OSP. The weight fraction of carbon should specify the total carbon content including this inorganic carbon.

If the new specie does not replace an old specie then the size of the arrays holding the property variables needs to be increased to accommodate the addition of the new specie. This is done by increasing the value of the NCOMP_GAS, NCOMP_LIQ, or NCOMP_SOLID parameter in the "osp_params.i" file. Since stoichiometric defaults and kinetic expressions are keyed to species sequence numbers extreme care should be taken in reordering the species list. If the list must be reordered then the coding in the "reaction.f" file should be carefully reviewed and updated. Additionally if the solid species list is reordered then the "attrition.f" file should be reviewed and updated.

A new species can be used by many of the OSP modules which do simple splitting and merging operations. However, if the new specie is to participate in chemical reactions then, as a minimum, stoichiometric coefficients must be defined which will probably involve the addition of new reactions (see below).

ADDING A NEW AUXILIARY SOLID PROPERTY

A new auxiliary solid property can be added by increasing the value of MAX_PROP_SOLID parameter in the "osp_params.i" file and defining an appropriate label in the "strm_initialize" subroutine. Coding within OSP will handle the passing of new properties from input to output streams and will combine streams with different values using rules equivalent to those used in computing compositions. Of course, a new property will only have an influence on the course of computations if new coding is added which makes explicit use of this property variable.

ADDING A NEW CHEMICAL REACTION

All definition and computations associated with chemical reactions available in OSP are collected in the "reaction.f" source code file. If an entire new set of reactions is desired, then a complete update of the subroutines RATE_SETUP and RATE_GEN are required and new reaction names need to be put into the REACTION_NAME initialization in the block data routine. No modification of currently implemented OSP unit operation modules is necessary since, to date, they all make use of this RATE_GEN subroutine to define required reaction kinetics and stoichiometric definitions are obtained from the G_STOICH, L_STOICH and S_STOICH arrays. No explicit knowledge about the details of a reaction are made by the modules.

To simply add a single new reaction the appropriate portions of the "reaction.f" file need to be modified. If a new reaction is added and is intended to be used only by the STOICH_REACT module then no kinetic model is required. In this case the name needs to be added to the REACTION_NAME initialization and the stoichiometric coefficients need to be defined in the RATE_SETUP routine. Finally, the parameter defining the number of reactions needs to be incremented. This is the NREACT parameter located in the "osp_rate.i" include file.

If a reaction is added in which a kinetic expression is required then the RATE_GEN routine needs the proper coding to allow rates to be computed. In its current form the modules which make use of the computed rate expressions assume that all reactions involve some component of a solid stream. That is they assume the rates are given then on a rate per particle basis. Any newly defined rate needs to compute the required rate on this basis. This does not mean any new rate that is added must necessarily explicitly contain a solid reactant. All that is required is that the kinetic rate expression is cast in such a fashion such the rate can be reported on a per particle basis.

All of the reactions considered in the OSP code are identified by a reference number and a name. For example, reaction number 10 is the decomposition of calcite, and has been given the name "CAL DECOMP". The reaction referenced number is then used in by RATE_GEN and RATE_SETUP_STOICH to identify the reaction rate constant and the stoichiometric coefficients. The number is also the index used in defining the reaction name in the variable array REACTION_NAME. A list of the reactions used in the OSP code at this time are given in the "Oil Shale Process Model (OSP) User's Manual."

The intent of the RATE_GEN routine is to define reaction kinetics which can be used by any module which requires them.. However, it is possible for any individual module to internally define its own reaction with appropriate stoichiometric and rate kinetic expressions and use these during its computations. It is not necessary for the OSP control routine to know the anything about the set of reactions used or potentially used in a module.

Steps to Add an Additional Chemical Reaction

The OSP code can model several reactions. The list of these reactions is in the "Oil Shale Process Model (OSP) User's Manual." A user / programmer can add a new reaction in a straight forward way. The process of adding a new reaction consists of the steps listed below.

The user should add any new species, that are reactants or products of the new reaction, to the existing species data base, as described in the previous subsection.

Next, add the reaction rate calculation into subroutine RATE_GEN. Any multiple resistance effects such as boundary layer and diffusional resistances must also be included in the reaction rate calculation.

The user should then add the reaction stoichiometric coefficients for all of the solid, liquid, and gas reactants and products, using the units described above, into subroutine RATE_SETUP_STOICH. The reaction name should be defined in the "block data" statement in the "service.f" file.

Finally, if necessary, the parameter NREACT in the "osp_rate.i" file should be increased.

SERVICE ROUTINES

Within the OSP code set there are a number of routines which are either essential to development of a new module or can be of great assistance. These routines are briefly described below. The programmer is referred to the source code for details of operation and input/output. Those routines which would be essential to a new module include strm_map, which handles the mapping of stream names to an internal index, and get_next_key which handles input file parsing.

Routines in "service.f" File

ATOM_BAL_OVERALL - Perform an atomic balance on streams of a selected module.

```
SUBROUTINE ATOM_BAL_OVERALL(SEQ, CIN, COUT, HIN, HOUT,
    OIN, OOUT,NIN, NOUT, SIN, SOUT)
```

BAL_OVERALL - Perform and print results of atomic and heat balance results for streams of a selected module.

```
SUBROUTINE BAL_OVERALL(SEQ)
```

BREAK_FILE - Detect the presence of a break request.

```
INTEGER FUNCTION BREAK_FILE(NAME_OF_BREAK_FILE,LEVEL)
```

BLOCK_A_INPUT - Controls reading of Block A input parameters.

```
LOGICAL FUNCTION INPUT_BLOCK_A_PARAMS(XRFLAG,
    XDIAM,XHEIGHT, XNS_AD, XDEL_COKE,_COMB, XDIL_MV,
    XDIL_MW, XPRAN, XIBED, XIDULIUTE, XDEBUG)
```

BLOCK_B_INPUT - Controls reading of Block B input parameters.

```
LOGICAL FUNCTION WRITE_BLOCK_B (XN_PDIST, MAXD,
    XVF_PDIST, XGRP_PDIST, XN_GRP_PDIST)
```

CHECK_BLOCK_A - Do some simple checks on validity of Block A input parameters.

```
INTEGER FUNCTION  CHECK_BLOCK_A (XRFLAG, XDIAM,
    XHEIGHT, XNS_AD, XDEL_COKE,_COMB, XDIL_MV,
    XDIL_MW, XPRAN, XIBED, XIDULIUTE, XDEBUG)
```

CHECK_BLOCK_B - Do some simple checks on validity of Block B input parameters.

```
INTEGER FUNCTION CHECK_BLOCK_B (XN_PDIST,
    MAXD,XVF_PDIST, XGRP_PDIST, XN_GRP_PDIST)
```

CONNECTIONS - Write a table of module connections to the ASCII output file.
INTEGER FUNCTION CONNECTIONS(SEQ)

DOTP - Compute dot product of two real*8 vectors.
REAL*8 FUNCTION DOTP (A,B,N)

DOTP2 - Compute dot product of a vector and n'th row of array.
REAL*8 FUNCTION DOTP2 (A,B,N,FIRST)

EQUAL - Compare two character strings.
LOGICAL FUNCTION EQUAL(LINE,STRING)

FIND_SPECIES_INDEX - Find the internal index of a species.
INTEGER FUNCTION
FIND_SPECIES_INDEX(NAME_IN,TYPE_IN,INDX,IER)

FIND_REACTION_INDEX - Find the internal index of a reaction.
INTEGER FUNCTION
FIND_REACTION_INDEX(NAME_IN,INDX,IER)

GET_INDX_CHECK - Obtain the value inside () in an ASCII line.
INTEGER FUNCTION GET_INDX_CHECK(LINE,MAX,INDX,IER)

GET_INDX2_CHECK - Obtain two comma delimited value inside () in an ASCII line.
INTEGER FUNCTION GET_INDX2_CHECK(LINE,MAX,MAX2,
INDX,INDX2,IER)

GET_NEXT_KEY - Parse a line for input values.
INTEGER FUNCTION GET_NEXT_KEY(CLINE,LOC,KEY,
VALUE,VALUE_FLAG,FIRST,LAST)

HEAT_BAL_OVERALL - Perform a heat balance on streams of a selected module.
SUBROUTINE HEAT_BAL_OVERALL(SEQ,IN,OUT)

HTX_WALL - Compute gas wall heat transfer coefficient in a pipe.
SUBROUTINE HTX_WALL(VEL, DIAM, RHO, VISC, CP, PRANDTL,
GASMW,HWALL)

HTX - Compute gas to particle surface heat transfer coefficient.
SUBROUTINE
HTX(DPA,VEL,RHO,VISC,CP,GASMW,PRANDTL,IBED,HTXC)

MOD_ERROR_LOG - Write an error message from a computational module to the selected logical unit.

```
SUBROUTINE MOD_ERROR_LOG(LU,SEQ,ROUTINE,IER)
```

MOVE - Move real*8 data from source to sink vector.

```
SUBROUTINE MOVE(A,B,N)
```

OUTP_BED_RESTIMES - Output a table of residence times.

```
SUBROUTINE OUTP_BED_RESTIMES (NPARTS, NPDIST,
    FIRST_INDEX, GRP_PDIST, N_GRP_PDIST, VF_PDIST, FLOW,
    VOL_PART, DENSITY, DIAMETER, POROSITY, VOLUME,
    N_PART, RES_TIME)
```

OUTP_GEN - Print stream information associated with a module.

```
SUBROUTINE OUTP_GEN(SEQ,TYPE,GFLAG,SFLAG,LFLAG)
```

PARSE - Parse a line for blank delimited character sequences.

```
INTEGER FUNCTION PARSE(LINE,LOC,ELEMENT)
```

PARTICLE_DENSITY - Determine particle density and residence time for each particle class.

```
SUBROUTINE PARTICLE_DENSITY (N_PDIST, FIRST_INDEX,
    GRP_PDIST, N_GRP_PDIST, VF_PDIST, FLOW, VOL_PART,
    DENSITY, POROSITY, VOLUME, N_PART, RES_TIME)
```

STREAM_COPY - Copy all stream variable values from one stream to another.

```
SUBROUTINE STREAM_COPY(TYPE, SOURCE, SINK, IER)
```

STRM_MAP - Obtain an internal index number for a stream name.

```
INTEGER FUNCTION STRM_MAP (TYPE, NAME, STRM_NUM,
    NEW, IER)
```

SUMF - Sum up elements of a real*8 vector.

```
REAL*8 FUNCTION SUMF (A,N)
```

SUMI - Sum up elements of a integer*4 vector.

```
INTEGER FUNCTION SUMF (A,N)
```

WRITE_BLOCK_A_PARAMS - Write table of Block A input parameters.

```
SUBROUTINE WRITE_BLOCK_A_PARAMS(XRFLAG, XDIAM,
    XHEIGHT, XNS_AD, XDEL_COKE,_COMB, XDIL_MV,
    XDIL_MW, XPRAN, XIBED, XIDULIUTE, XDEBUG)
```

WRITE_BLOCK_B_PARAMS - Write table of Block B input parameters.

```
SUBROUTINE WRITE_BLOCK_B_PARAMS(XN_PDIST, MAXD,
    XVF_PDIST, XGRP_PDIST, XN_GRP_PDIST)
```

WRITE_FORM - Write form feed to specified logical unit.
SUBROUTINE WRITE_FORM_FEED(LU)

WRITE_MOD_HEADER - Write module header information to ASCII output file.
SUBROUTINE WRITE_MOD_HEADER(SEQ)

WRITE_VAR_TABLE - Write table of stream variables and flagged species.
SUBROUTINE WRITE_VAR_TABLE(GAS,LIQ,SOLID,NPARTS)

Routines in "props.f" File

COMBUST - Compute stoichiometry for complete combustion of a solid.
SUBROUTINE COMBUST (WF_C,WF_H,WF_O,WF_N,WF_S,AO2,
ACO2,AH2O,ANO2,ASO2)

DIFFUSIVITY - Compute diffusivity of a selected gas.
SUBROUTINE DIFFUSIVITY (T,P,GAS,DIL_MV,DIL_MW,DIFF)

GAS_CP - Compute gas species heat capacities at a given temperature.
SUBROUTINE GAS_CPF(T)

GAS_ENTH - Compute gas species enthalpies at a given temperature.
SUBROUTINE GAS_ENTHF(T)

GAS_MWF - Compute and set the gas species molecular weights.
SUBROUTINE GAS_MWF

GAS_VISCF - Compute gas species viscosities at a given temperature.
SUBROUTINE GAS_VISCF(T)

LIQ_MWF - Compute and set the liquid species molecular weights.
SUBROUTINE LIQ_MWF

LIQ_CPF - Compute liquid species heat capacities at a given temperature.
SUBROUTINE LIQ_CPF(T)

LIQ_ENTH - Compute liquid species enthalpies at a given temperature.
SUBROUTINE LIQ_ENTHF(T)

POR_DENF - Compute porosity and density of a given solid stream.
SUBROUTINE POR_DENF(MASS_FLOW, PARTICLE_FLOW,
PARTICLE_VOLUME,WEIGHT_FRACTIONS,POROSITY,
DENSITY)

SOLID_ENTH - Compute solid species enthalpies at a given temperature.
SUBROUTINE SOLID_CPF(T)

SOLID_CP - Compute solid species heat capacities at a given temperature.
SUBROUTINE SOLID_CPF(T)

VISCF - Compute viscosity of a gas mixture.
SUBROUTINE VISCF(T,Y,VISC)

Routines in "reaction.f" File

RATE_SETUP - Define stoichiometric coefficients for rate_gen.
SUBROUTINE RATE_SETUP_STOICH(IER)

RATE_MTX_COEF - Compute gas to particle surface mass transfer coefficient.
SUBROUTINE RATE_MTX_COEF (DIAM_SOLID,RHO,VEL,VISC,
DIFF,IBED,KMASS)

RATE_DIFFE - Compute effective gas diffusivities.
SUBROUTINE RATE_DIFFE(DIFF,POROSITY,DIFFE)

RATE_GEN - Compute reaction rates for a given state of system.
SUBROUTINE RATE_GEN(RHOG, VISC, C, REL_LIQ,VELREL, TG, TS,
P, Y_SOLID,DEN_SOLID, DIAM_SOLID, VOL_SOLID,
POR_SOLID,DIL_MV,DIL_MW,RHO_COMB_ORIG,
RHO_KER_ORIG,RHO_FES2_ORIG,NS_ADS, IBED,IDILUTE,
KMASS_IN_O2,KMASS_IN_OIL,DEL_COKE_FRAC,
DEL_COKE_COMB_FRAC, RATE)

Routines in "combine.f" File

COMBINE - Combine selected streams.
SUBROUTINE COMBINE_STREAMS(IER)

Routines in "plug.f" File

PLUG_FLOW - Perform calculations for a plug flow reactor.
SUBROUTINE PLUG_FLOW(IER)

Routines in "well_mix.f" File

WELL_MIX- Perform calculations for a well mixed reactor.
SUBROUTINE WELL_MIX(IER)

Routines in "binary_io.f" File

READ_WORDS - Read a binary record from an open file.
INTEGER FUNCTION READ_WORDS(LU, IBUF, NWORDS, IER)

WRITE_WORDS - Write a binary record to an open file.
INTEGER FUNCTION WRITE_WORDS(LU, IBUF, NWORDS, IER)

SPECIFIC MACHINE DEPENDENCIES IN THE OSP CODE

The OSP code was developed on HP-9000-340, 375, and 730 work stations. It has also been implemented on a CRAY-XMP computer. The code is written primarily in FORTRAN-77 and has few machine dependencies. A few C subroutines are used for string processing and are located in the "string_cray.f" file. In this section, a list and description is provided of the dependencies that were identified when the code was implemented on the CRAY-XMP. If the code is to be moved to another machine or a different operating system, the code developer should provide coding substitutes appropriate for his system.

A UNIX script called "osptocray" and a SED input file was developed to aid in the code conversion. The script uses the SED file to modify each subroutine that make up OSP. The modified subroutines are put into a sub-directory called "osp_cray". The SED input file changes the code in 5 ways. References to a FORTRAN subroutine %LOC on the HP-9000 is changed to LOC. The path to the include files is changed from "/usr/local/osp" to "..". References to 4 byte words is changed to single byte words. And two subroutines that use double precision variables are changed to routines that use single precision ones.

Four subroutines which are used by the OSP code that are in a public library on our systems are included. These files are machine_cray.f, math_cray.f, string_cray.f, and hmp_cray.f,. The subroutines used by OSP which are in the "machine_cray.f" file are RUNTIME, RUNNINGTIME, and DATETIME. The subroutine used by OSP in the "math_cray.f" file is SORT. Most of the subroutines in the "string_cray.f" file are used by the OSP code. Some of the subroutines in "string_cray.f" are written in C. The subroutines in the "hmp_cray.f" file are not needed by a user of OSP unless the user plans to use the data manipulation and graphics display code, DSP. Dummy subroutines can be provided for the subroutines in the "hmp_cray.f" file if the user does not plan to use DSP.

Precision Options for LSODE

The subroutine set LSODE, used in the computations, has single precision and a double precision versions. The double precision version is appropriate for most machines, however the single precision version should be used on CRAY machines. The single precision versions for the subroutines used by LSODE are located in the "xsubs_cray.f" file.

REFERENCES

Thorsness, C.B. and Aldis, D.F., "Oil Shale Process Model (OSP) Theory Manual", *Lawrence Livermore National Laboratory UCRL–MA-119226,* Dec. 1994.

Thorsness, C.B. and Aldis D.F. , "Oil Shale Process Model (OSP) User's Manual", *Lawrence Livermore National Laboratory UCRL–ID -119260,* 1995.

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551